

Lectures 2&3: Introduction to SQL

Announcements!

1. If you still have Jupyter trouble, let us know!
 - I bought a surface to try to help debug... 😊
2. New conflict policy with 144! Hope you can all take the class
 - Be nice to the course staff!
3. Problem Set #1 is released!
4. **Jeff is a TA**, and I am a jerk.



Lecture 2: SQL Part I

Today's Lecture

1. SQL introduction & schema definitions
 - ACTIVITY: Table creation
2. Basic single-table queries
 - ACTIVITY: Single-table queries!
3. Multi-table queries
 - ACTIVITY: Multi-table queries!

1. SQL Introduction & Definitions

What you will learn about in this section

1. What is SQL?
2. Basic schema definitions
3. Keys & constraints intro
4. ACTIVITY: CREATE TABLE statements

SQL Motivation

- Dark times 5 years ago.
 - Are databases dead?
- Now, as before: everyone sells SQL
 - Pig, Hive, Impala
- “Not-Yet-SQL?”



Basic SQL

SQL Introduction

- SQL is a standard language for querying and manipulating data
- SQL is a **very high-level** programming language
 - This works because it is optimized well!
- Many standards out there:
 - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3),
 - Vendors support various subsets

SQL stands for
Structured Query Language

NB: Probably the world's most successful **parallel**
programming language (multicore?)

SQL is a...

- Data Definition Language (DDL)
 - Define relational *schemata*
 - Create/alter/delete tables and their attributes
- Data Manipulation Language (DML)
 - Insert/delete/modify tuples in tables
 - Query one or more tables – discussed next!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

A relation or table is a multiset of tuples having the attributes specified by the schema

Let's break this definition down

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

A multiset is an unordered list (or: a set with multiple duplicate instances allowed)

List: [1, 1, 2, 3]

Set: {1, 2, 3}

Multiset: {1, 1, 2, 3}

i.e. no *next()*, etc. methods!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

An attribute (or column) is a typed data entry present in each tuple in the relation

*NB: Attributes must have an **atomic** type in standard SQL, i.e. not a list, set, etc.*

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

Also referred to sometimes as a record

A tuple or row is a single entry in the table having the attributes specified by the schema

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

The number of tuples is the cardinality of the relation

The number of attributes is the arity of the relation

Data Types in SQL

- Atomic types:
 - Characters: CHAR(20), VARCHAR(50)
 - Numbers: INT, BIGINT, SMALLINT, FLOAT
 - Others: MONEY, DATETIME, ...
- Every attribute must have an atomic type
 - Hence tables are flat

Table Schemas

- The **schema** of a table is the table name, its attributes, and their types:

```
Product(Pname: string, Price: float, Category:  
string, Manufacturer: string)
```

- A **key** is an attribute whose values are unique; we underline a key

```
Product(Pname: string, Price: float, Category:  
string, Manufacturer: string)
```

Key constraints

A key is a minimal subset of attributes that acts as a unique identifier for tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
 - i.e. if two tuples agree on the values of the key, then they must be the same tuple!

```
Students(sid:string, name:string, gpa: float)
```

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?

NULL and NOT NULL

- To say “don’t know the value” we use **NULL**
 - NULL has (sometimes painful) semantics, more detail later

```
Students(sid:string, name:string, gpa: float)
```

sid	name	gpa
123	Bob	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., “name” in this table

General Constraints

- We can actually specify arbitrary assertions
 - E.g. *“There cannot be 25 people in the DB class”*
- In practice, we don’t specify many such constraints. Why?
 - Performance!

Whenever we do something ugly (or avoid doing something convenient) it’s for the sake of performance

Summary of Schema Information

- Schema and Constraints are how databases understand the semantics (meaning) of data
- They are also useful for optimization
- SQL supports general constraints:
 - Keys and foreign keys are most important
 - We'll give you a chance to write the others

ACTIVITY: [Activity-2-1.ipynb](#)

2. Single-table queries

What you will learn about in this section

1. The SFW query
2. Other useful operators: LIKE, DISTINCT, ORDER BY
3. ACTIVITY: Single-table queries

SQL Query

- Basic form (there are many many more bells and whistles)

```
SELECT <attributes>  
FROM   <one or more relations>  
WHERE  <conditions>
```

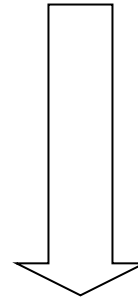
Call this a SFW query.

Simple SQL Query: Selection

Selection is the operation of filtering a relation's tuples on some condition

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT *  
FROM Product  
WHERE Category = 'Gadgets'
```



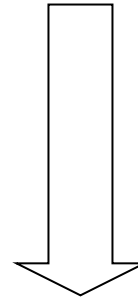
PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks

Simple SQL Query: Projection

Projection is the operation of producing an output table with tuples that have a subset of their prior attributes

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```



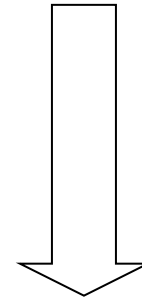
PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks

Notation

Input schema

Product(PName, Price, Category, Manufacturer)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```



Output schema

Answer(PName, Price, Manufacturer)

A Few Details

- SQL **commands** are case insensitive:
 - Same: SELECT, Select, select
 - Same: Product, product
- **Values** are **not**:
 - Different: 'Seattle', 'seattle'
- Use single quotes for constants:
 - 'abc' - yes
 - "abc" - no

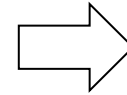
LIKE: Simple String Pattern Matching

```
SELECT *  
FROM Products  
WHERE PName LIKE '%gizmo%'
```

- `s LIKE p`: pattern matching on strings
- `p` may contain two special symbols:
 - `%` = any sequence of characters
 - `_` = any single character

DISTINCT: Eliminating Duplicates

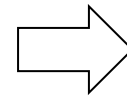
```
SELECT DISTINCT Category  
FROM Product
```



Category
Gadgets
Photography
Household

Versus

```
SELECT Category  
FROM Product
```



Category
Gadgets
Gadgets
Photography
Household

ORDER BY: Sorting the Results

```
SELECT PName, Price, Manufacturer
FROM Product
WHERE Category='gizmo' AND Price > 50
ORDER BY Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

ACTIVITY: [Activity-2-2.ipynb](#)

3. Multi-table queries

What you will learn about in this section

1. Foreign key constraints
2. Joins: basics
3. Joins: SQL semantics
4. **ACTIVITY: Multi-table queries**

Foreign Key constraints

- Suppose we have the following schema:

```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)
```

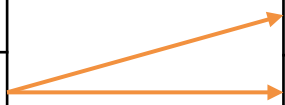
- And we want to impose the following constraint:
 - 'Only bona fide students may enroll in courses' i.e. a student must appear in the Students table to enroll in a class

Students

sid	name	gpa
101	Bob	3.2
123	Mary	3.8

Enrolled

student_id	cid	grade
123	564	A
123	537	A+



student_id alone is not a key- what is?

We say that student_id is a foreign key that refers to Students

Declaring Foreign Keys

```
Students(sid: string, name: string, gpa: float)  
Enrolled(student_id: string, cid: string, grade: string)
```

```
CREATE TABLE Enrolled(  
    student_id CHAR(20),  
    cid        CHAR(20),  
    grade      CHAR(10),  
    PRIMARY KEY (student_id, cid),  
    FOREIGN KEY (student_id) REFERENCES Students(sid)  
)
```

Foreign Keys and update operations

```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)
```

- What if we insert a tuple into Enrolled, but no corresponding student?
 - INSERT is rejected (foreign keys are constraints)!
- What if we delete a student?
 - 1. Disallow the delete
 - 2. Remove all of the courses for that student
 - 3. *SQL allows a third via NULL (not yet covered)*

DBA chooses (syntax in the book)

Keys and Foreign Keys

Company

<u>CName</u>	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a
foreign key vs.
a key here?

Product

<u>PName</u>	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

Joins

```
Product(PName, Price, Category, Manufacturer)  
Company(CName, StockPrice, Country)
```

Ex: Find all products under \$200 manufactured in Japan;
return their names and prices.

```
SELECT PName, Price  
FROM   Product, Company  
WHERE  Manufacturer = CName  
        AND Country='Japan'  
        AND Price <= 200
```

Note: we will often omit attribute types in schema definitions for brevity, but assume attributes are always atomic types

Joins

```
Product(PName, Price, Category, Manufacturer)
Company(CName, StockPrice, Country)
```

Ex: Find all products under \$200 manufactured in Japan;
return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
      AND Country='Japan'
      AND Price <= 200
```

A join between tables returns
all unique combinations of
their tuples **which meet**
some specified join condition

Joins

```
Product(PName, Price, Category, Manufacturer)  
Company(CName, StockPrice, Country)
```

Several equivalent ways to write a basic join in SQL:

```
SELECT PName, Price  
FROM   Product, Company  
WHERE  Manufacturer = CName  
       AND Country='Japan'  
       AND Price <= 200
```

```
SELECT PName, Price  
FROM   Product  
JOIN   Company ON Manufacturer = Cname  
              AND Country='Japan'  
WHERE  Price <= 200
```

A few more later on...

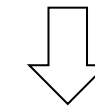
Joins

Product

PName	Price	Category	Manuf
Gizmo	\$19	Gadgets	GWorks
Powergizmo	\$29	Gadgets	GWorks
SingleTouch	\$149	Photography	Canon
MultiTouch	\$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan



```

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
      AND Country='Japan'
      AND Price <= 200
  
```

PName	Price
SingleTouch	\$149.99

Tuple Variable Ambiguity in Multi-Table

```
Person(name, address, worksfor)  
Company(name, address)
```

```
SELECT DISTINCT name, address  
FROM           Person, Company  
WHERE          worksfor = name
```

Which “address” does
this refer to?

Which “name”s??

Tuple Variable Ambiguity in Multi-Table

```
Person(name, address, worksfor)
Company(name, address)
```

Both equivalent
ways to resolve
variable
ambiguity

```
SELECT DISTINCT Person.name, Person.address
FROM           Person, Company
WHERE          Person.worksfor = Company.name
```

```
SELECT DISTINCT p.name, p.address
FROM           Person p, Company c
WHERE          p.worksfor = c.name
```

Meaning (Semantics) of SQL Queries

```

SELECT x1.a1, x1.a2, ..., xn.ak
FROM   R1 AS x1, R2 AS x2, ..., Rn AS xn
WHERE  Conditions(x1, ..., xn)
  
```

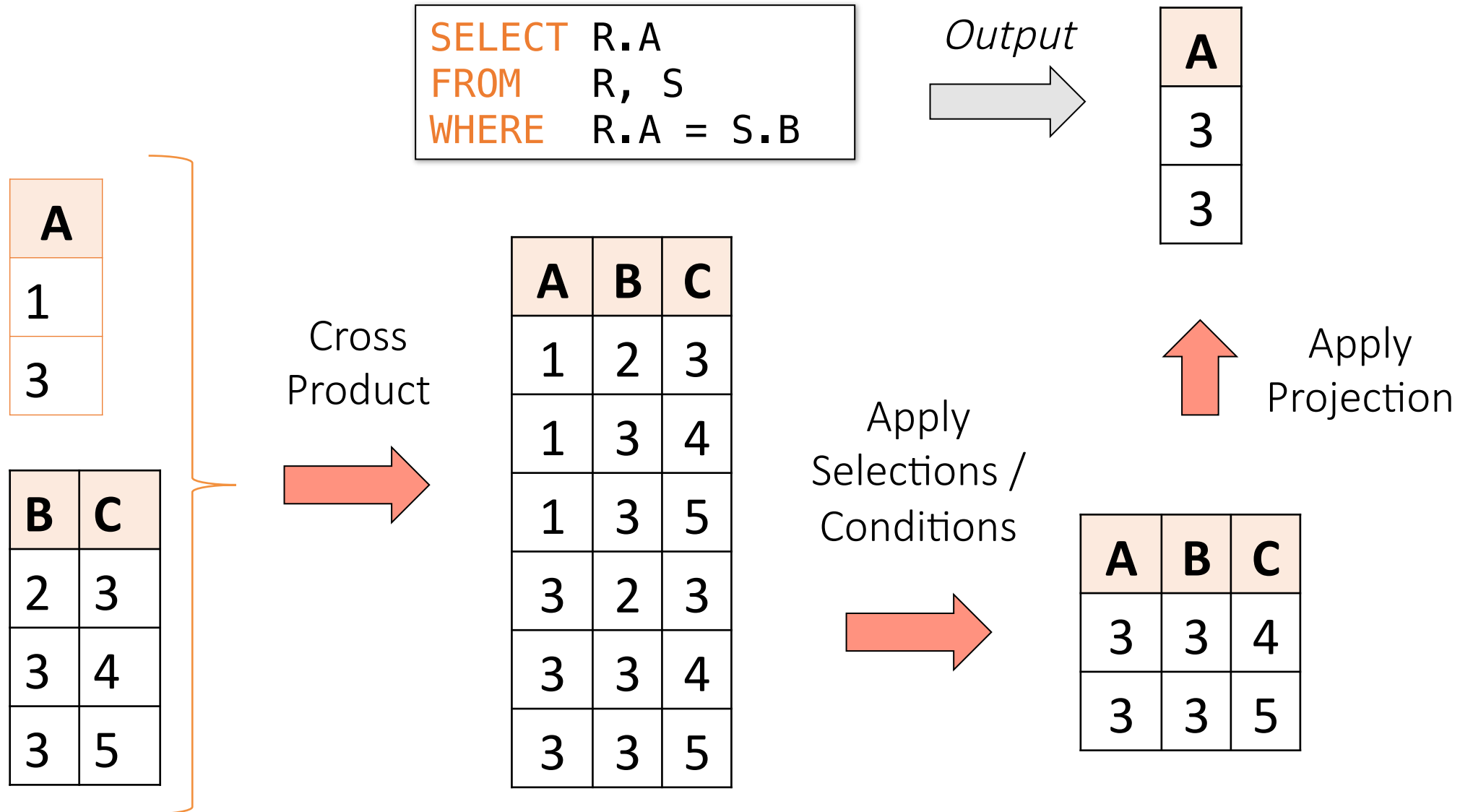
Almost never the *fastest* way to compute it!

```

Answer = {}
for x1 in R1 do
  for x2 in R2 do
    ....
    for xn in Rn do
      if Conditions(x1, ..., xn)
        then Answer = Answer ∪ {x1.a1, x1.a2, ..., xn.ak}
return Answer
  
```

Note: this is a *multiset* union

An example of SQL semantics



Note the *semantics* of a join

```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

1. Take **cross product**:

$$X = R \times S$$

Recall: Cross product ($A \times B$) is the set of all unique tuples in A, B

Ex: $\{a, b, c\} \times \{1, 2\}$
 $= \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$

2. Apply **selections / conditions**:

$$Y = \{(r, s) \in X \mid r.A = s.B\}$$

= Filtering!

3. Apply **projections** to get final output:

$$Z = (y.A) \text{ for } y \in Y$$

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on...)

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not actually how the DBMS executes it under the covers

A Subtlety about Joins

```
Product(PName, Price, Category, Manufacturer)  
Company(CName, StockPrice, Country)
```

Find all countries that manufacture some product in the 'Gadgets' category.

```
SELECT Country  
FROM   Product, Company  
WHERE  Manufacturer=CName AND Category='Gadgets'
```

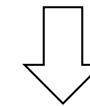
A subtlety about Joins

Product

PName	Price	Category	Manuf
Gizmo	\$19	Gadgets	GWorks
Powergizmo	\$29	Gadgets	GWorks
SingleTouch	\$149	Photography	Canon
MultiTouch	\$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan



```

SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
      AND Category='Gadgets'
  
```

Country
?
?

What is the problem ?
What's the solution ?

ACTIVITY: [Lecture-2-3.ipynb](#)

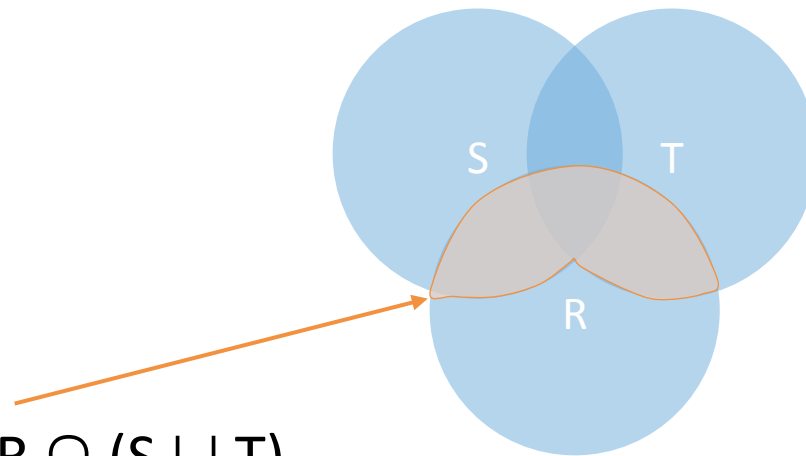
An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```

What does it compute?

An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```



Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Go back to the semantics!

An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```

- Recall the semantics!
 1. Take cross-product
 2. Apply selections / conditions
 3. Apply projection
- If $S = \{\}$, then the cross product of $R, S, T = \{\}$, and the query result = $\{\}$!

Must consider semantics here.
Are there more explicit way to do set operations like this?

Lecture 3: SQL Part II

Announcements

1. I hope you are setup on Jupyter
 - We're posting a piazza post about Python help
 - We're also trying to get some windows tips together.
2. Project 1 will be released on Thursday

Today's Lecture

1. Set operators & nested queries
 - ACTIVITY: Set operator subtleties
2. Aggregation & GROUP BY
 - ACTIVITY: Fancy SQL Part I
3. Advanced SQL-izing
 - ACTIVITY: Fancy SQL Part II

1. Set Operators & Nested Queries

What you will learn about in this section

1. Multiset operators in SQL
2. Nested queries
3. ACTIVITY: Set operator subtleties

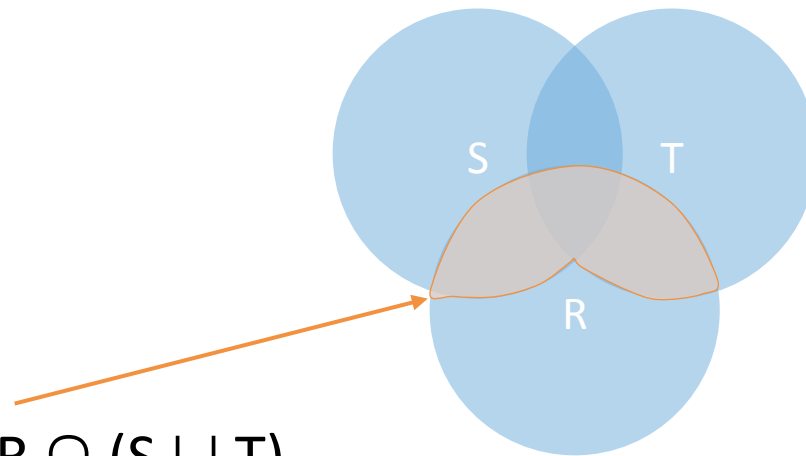
An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```

What does it compute?

An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```



Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Go back to the semantics!

An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```

- Recall the semantics!
 1. Take cross-product
 2. Apply selections / conditions
 3. Apply projection
- If $S = \{\}$, then the cross product of $R, S, T = \{\}$, and the query result = $\{\}$!

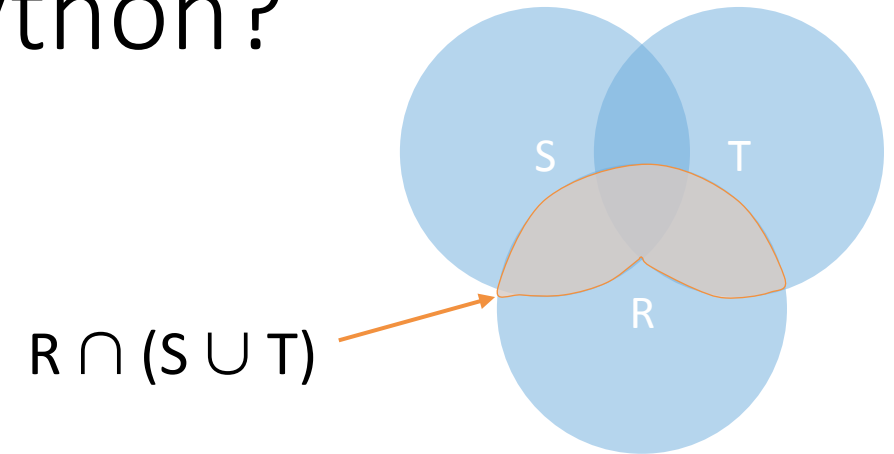
Must consider semantics here.
Are there more explicit way to do set operations like this?

What does this look like in Python?

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```

- Semantics:

1. Take cross-product
2. Apply selections / conditions
3. Apply projection

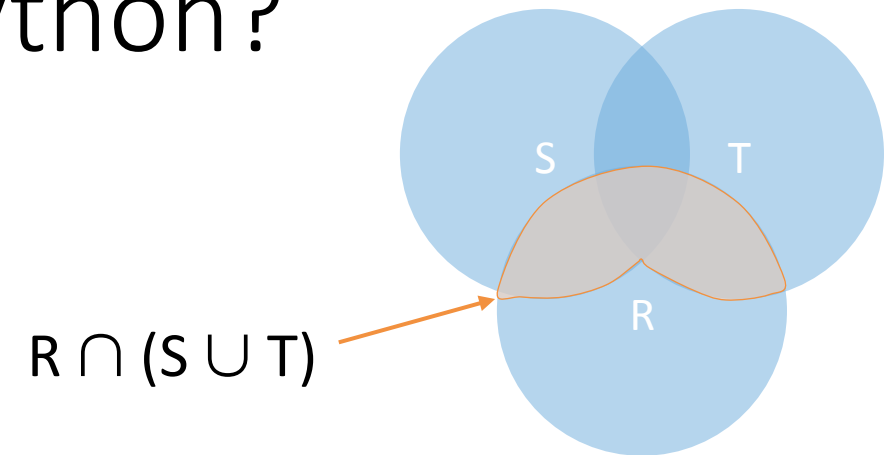


Joins / cross-products are just **nested for loops** (in simplest implementation)!

If-then statements!

What does this look like in Python?

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```



```
output = {}  
  
for r in R:  
    for s in S:  
        for t in T:  
            if r['A'] == s['A'] or r['A'] == t['A']:  
                output.add(r['A'])  
return list(output)
```

Can you see now what happens if $S = []$?

See bonus activity on website!

Multiset Operations

Recall Multisets

Multiset X

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)



Equivalent
Representations
of a Multiset

$\lambda(X)$ = "Count of tuple in X"
(Items not listed have
implicit count 0)

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

*Note: In a set all
counts are {0,1}.*

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

 $\cap$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

 $=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0

$$\lambda(Z) = \min(\lambda(X), \lambda(Y))$$

For sets, this is
intersection

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

 $\cup$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

 $=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

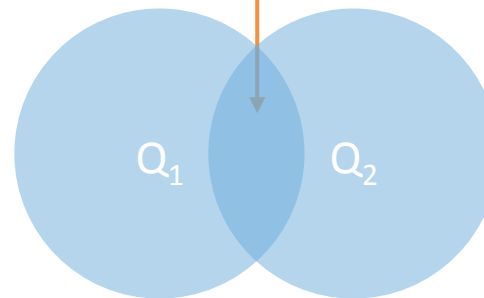
For sets,
this is union

Multiset Operations in SQL

Explicit Set Operators: INTERSECT

```
SELECT R.A  
FROM   R, S  
WHERE  R.A=S.A  
INTERSECT  
SELECT R.A  
FROM   R, T  
WHERE  R.A=T.A
```

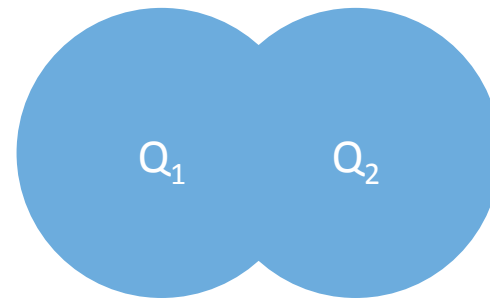
$$r.A \mid r.A=s.A \cap \{r.A \mid r.A=t.A\}$$



UNION

```
SELECT  R.A  
FROM    R, S  
WHERE   R.A=S.A  
UNION  
SELECT  R.A  
FROM    R, T  
WHERE   R.A=T.A
```

$$r.A \mid r.A=s.A \cup \{r.A \mid r.A=t.A\}$$



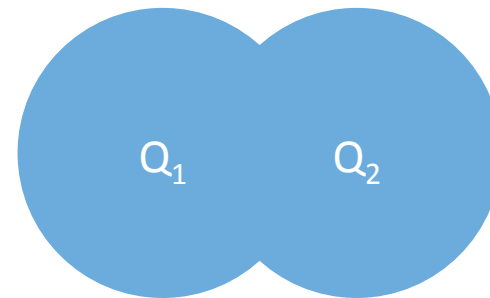
Why aren't there duplicates?

What if we want duplicates?

UNION ALL

```
SELECT  R.A  
FROM    R, S  
WHERE   R.A=S.A  
UNION ALL  
SELECT  R.A  
FROM    R, T  
WHERE   R.A=T.A
```

$$r.A \mid r.A=s.A \cup \{r.A \mid r.A=t.A\}$$



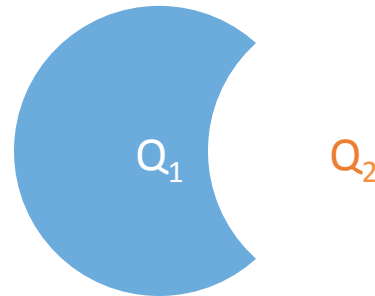
*ALL indicates
Multiset
operations*

EXCEPT

```

SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A

```

$$r.A \mid r.A=s.A \setminus \{r.A \mid r.A=t.A\}$$


What is the multiset version?

INTERSECT: Still some subtle problems...

```
Company(name, hq_city)  
Product(pname, maker, factory_loc)
```

```
SELECT hq_city  
FROM   Company, Product  
WHERE  maker = name  
        AND factory_loc = 'US'  
  
INTERSECT  
SELECT hq_city  
FROM   Company, Product  
WHERE  maker = name  
        AND factory_loc = 'China'
```

*“Headquarters of
companies which
make gizmos in US
AND China”*

What if two companies have HQ in US: BUT one has factory in China (but not US) and vice versa? **What goes wrong?**

INTERSECT: Remember the semantics!

```
Company(name, hq_city) AS C
Product(pname, maker,
factory_loc) AS P
```

```
SELECT hq_city
FROM Company, Product
WHERE maker = name
AND factory_loc='US'
```

```
INTERSECT
SELECT hq_city
FROM Company, Product
WHERE maker = name
AND factory_loc='China'
```

Example: C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

INTERSECT: Remember the semantics!

```
Company(name, hq_city) AS C
Product(pname, maker,
factory_loc) AS P
```

```
SELECT hq_city
FROM Company, Product
WHERE maker = name
AND factory_loc='US'
```

```
INTERSECT
SELECT hq_city
FROM Company, Product
WHERE maker = name
AND factory_loc='China'
```

Example: C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

X Co has a factory in the US (but not China)
Y Inc. has a factor in China (but not US)

But Seattle is returned by the query!

We did the INTERSECT
on the wrong attributes!

One Solution: Nested Queries

```
Company(name, hq_city)
Product(pname, maker, factory_loc)
```

```
SELECT DISTINCT hq_city
FROM   Company, Product
WHERE  maker = name
      AND name IN (
          SELECT maker
          FROM   Product
          WHERE  factory_loc = 'US')
      AND name IN (
          SELECT maker
          FROM   Product
          WHERE  factory_loc = 'China')
```

*“Headquarters of companies which make gizmos in US **AND** China”*

Note: If we hadn't used DISTINCT here, how many copies of each hq_city would have been returned?

High-level note on nested queries

- We can do nested queries because SQL is ***compositional***:
 - Everything (inputs / outputs) is represented as multisets- the output of one query can thus be used as the input to another (nesting)!
- This is extremely powerful!

Nested queries: Sub-queries Returning Relations

Another
example:

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

```
SELECT c.city
FROM   Company c
WHERE  c.name IN (
    SELECT pr.maker
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
    AND    p.buyer = 'Joe Blow')
```

“Cities where one
can find
companies that
manufacture
products bought
by Joe Blow”

Nested Queries

Is this query equivalent?

```
SELECT c.city
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.maker
      AND pr.name = p.product
      AND p.buyer = 'Joe Blow'
```

Beware of duplicates!

Nested Queries

```
SELECT DISTINCT c.city
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.maker
      AND pr.name = p.product
      AND p.buyer = 'Joe Blow'
```

```
SELECT DISTINCT c.city
FROM   Company c
WHERE  c.name IN (
    SELECT pr.maker
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
          AND p.buyer = 'Joe Blow')
```

Now they are equivalent

Subqueries Returning Relations

You can also use operations of the form:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- $\text{EXISTS } R$

ANY and ALL not supported by SQLite.

Ex:

`Product(name, price, category, maker)`

```
SELECT name
FROM   Product
WHERE  price > ALL(
        SELECT price
        FROM   Product
        WHERE  maker = 'Gizmo-Works')
```

Find products that
are more expensive
than all those
produced by
“Gizmo-Works”

Subqueries Returning Relations

You can also use operations of the form:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- EXISTS R

Ex:

`Product(name, price, category, maker)`

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = 'Gizmo-Works'
      AND EXISTS(
        SELECT p2.name
        FROM   Product p2
        WHERE  p2.maker <> 'Gizmo-Works'
              AND p1.name = p2.name)
```

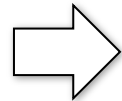
<> means !=

Find 'copycat' products, i.e. products made by competitors with the same names as products made by "Gizmo-Works"

Nested queries as alternatives to INTERSECT and EXCEPT

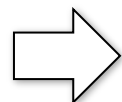
INTERSECT and EXCEPT not in some DBMSs!

```
(SELECT R.A, R.B
 FROM R)
INTERSECT
(SELECT S.A, S.B
 FROM S)
```



```
SELECT R.A, R.B
FROM R
WHERE EXISTS (
  SELECT *
  FROM S
  WHERE R.A=S.A AND R.B=S.B)
```

```
(SELECT R.A, R.B
 FROM R)
EXCEPT
(SELECT S.A, S.B
 FROM S)
```



```
SELECT R.A, R.B
FROM R
WHERE NOT EXISTS (
  SELECT *
  FROM S
  WHERE R.A=S.A AND R.B=S.B)
```

If R, S have no duplicates, then can write without sub-queries (HOW?)

A question for Database Fans & Friends

- Can we express the previous nested queries as single SFW queries?
- Hint: show that all SFW queries are **monotone** (roughly: more tuples, more answers).
 - A query with **ALL** is often not monotone

Correlated Queries

```
Movie(title, year, director, length)
```

```
SELECT DISTINCT title
FROM   Movie AS m
WHERE  year <> ANY(
        SELECT year
        FROM   Movie
        WHERE  title = m.title)
```

Find movies whose title appears more than once.

Note the scoping of the variables!

Note also: this can still be expressed as single SFW query...

Complex Correlated Query

```
Product(name, price, category, maker, year)
```

```
SELECT DISTINCT x.name, x.maker
FROM   Product AS x
WHERE  x.price > ALL(
        SELECT y.price
        FROM   Product AS y
        WHERE  x.maker = y.maker
              AND y.year < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Can be very powerful (also much harder to optimize)

Basic SQL Summary

- SQL provides a high-level declarative language for manipulating data (DML)
- The workhorse is the SFW block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed.

Activity-3-1.ipynb

2. Aggregation & GROUP BY

What you will learn about in this section

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics
4. ACTIVITY: Fancy SQL Pt. I

Aggregation

```
SELECT AVG(price)
FROM Product
WHERE maker = "Toyota"
```

```
SELECT COUNT(*)
FROM Product
WHERE year > 1995
```

- SQL supports several **aggregation** operations:
 - SUM, COUNT, MIN, MAX, AVG

Except COUNT, all aggregations apply to a single attribute

Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM Product
WHERE year > 1995
```

Note: Same as COUNT().
Why?*

We probably want:

```
SELECT COUNT(DISTINCT category)
FROM Product
WHERE year > 1995
```

More Examples

```
Purchase(product, date, price, quantity)
```

```
SELECT SUM(price * quantity)
FROM   Purchase
```

```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

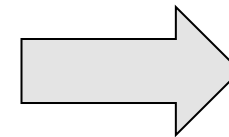
What do these mean?

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```



50 (= 1*20 + 1.50*20)

Grouping and Aggregation

```
Purchase(product, date, price, quantity)
```

```
SELECT    product,  
          SUM(price * quantity) AS TotalSales  
FROM      Purchase  
WHERE     date > '10/1/2005'  
GROUP BY product
```

Find total sales
after 10/1/2005
per product.

Let's see what this means...

Grouping and Aggregation

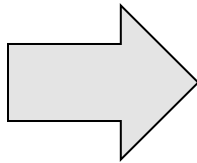
Semantics of the query:

1. Compute the **FROM** and **WHERE** clauses
2. Group by the attributes in the **GROUP BY**
3. Compute the **SELECT** clause: grouped attributes and aggregates

1. Compute the FROM and WHERE clauses

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
```

FROM



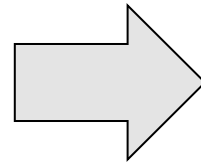
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY



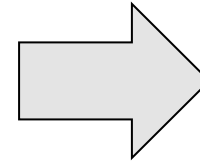
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the **SELECT** clause: grouped attributes and aggregates

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT



Product	TotalSales
Bagel	50
Banana	15

GROUP BY v.s. Nested Quereis

```
SELECT    product, Sum(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
```

```
SELECT DISTINCT x.product,
               (SELECT Sum(y.price*y.quantity)
                FROM   Purchase y
                WHERE  x.product = y.product
                   AND y.date > '10/1/2005') AS TotalSales
FROM      Purchase x
WHERE     x.date > '10/1/2005'
```

HAVING Clause

```
SELECT    product, SUM(price*quantity)
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
HAVING    SUM(quantity) > 100
```

HAVING clauses contains conditions on aggregates

Whereas WHERE clauses condition on individual tuples...

Same query as before, except that we consider only products that have more than 100 buyers

General form of Grouping and Aggregation

SELECT	S
FROM	$R_1, \dots, R_n$
WHERE	C_1
GROUP BY	$a_1, \dots, a_k$
HAVING	C_2

Why?

- S = Can ONLY contain attributes $a_1, \dots, a_k$ and/or aggregates over other attributes
- C_1 = is any condition on the attributes in $R_1, \dots, R_n$
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

SELECT	S
FROM	$R_1, \dots, R_n$
WHERE	C_1
GROUP BY	$a_1, \dots, a_k$
HAVING	C_2

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition C_1 on the attributes in $R_1, \dots, R_n$
2. **GROUP BY** the attributes $a_1, \dots, a_k$
3. **Apply condition C_2 to each group (may have aggregates)**
4. Compute aggregates in S and return the result

Group-by v.s. Nested Query

```
Author(login, name)
Wrote(login, url)
```

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM   Author
WHERE  COUNT(
        SELECT Wrote.url
        FROM   Wrote
        WHERE  Author.login = Wrote.login) > 10
```

This is
SQL by
a novice

Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name  
FROM Author, Wrote  
WHERE Author.login = Wrote.login  
GROUP BY Author.name  
HAVING COUNT(Wrote.url) > 10
```

This is
SQL by
an expert

No need for **DISTINCT**: automatically from **GROUP BY**

Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- *With nested*: How many times do we do a SFW query over all of the Wrote relations?
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be much more efficient!

Activity-3-2.ipynb

3. Advanced SQL-izing

What you will learn about in this section

1. Quantifiers
2. NULLs
3. Outer Joins
4. ACTIVITY: Fancy SQL Pt. II

Quantifiers

```
Product(name, price, company)
Company(name, city)
```

```
SELECT DISTINCT Company.cname
FROM    Company, Product
WHERE   Company.name = Product.company
        AND Product.price < 100
```

Find all companies
that make some
products with price
< 100

An existential quantifier is a
logical quantifier (roughly)
of the form “there exists”

Existential: easy ! 😊

Quantifiers

```
Product(name, price, company)
Company(name, city)
```

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.name NOT IN(
    SELECT Product.company
    FROM Product.price >= 100)
```

A universal quantifier is of the form “for all”

Find all companies with products all having price < 100



Equivalent

Find all companies that make only products with price < 100

Universal: hard ! ☹️

NULLS in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
 - Value does not exist
 - Value exists but is unknown
 - Value not applicable
 - Etc.
- The schema specifies for each attribute if it can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs?

Null Values

- *For numerical operations, NULL -> NULL:*
 - If $x = \text{NULL}$ then $4 \cdot (3 - x) / 7$ is still NULL
- *For boolean operations, in SQL there are three values:*

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

- If $x = \text{NULL}$ then $x = \text{"Joe"}$ is UNKNOWN

Null Values

- $C1 \text{ AND } C2 = \min(C1, C2)$
- $C1 \text{ OR } C2 = \max(C1, C2)$
- $\text{NOT } C1 = 1 - C1$

```
SELECT *  
FROM Person  
WHERE (age < 25)  
      AND (height > 6 AND weight > 190)
```

Won't return e.g.
(age=20
height=NULL
weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

Null Values

Unexpected behavior:

```
SELECT *  
FROM Person  
WHERE age < 25 OR age >= 25
```

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *  
FROM Person  
WHERE age < 25 OR age >= 25  
      OR age IS NULL
```

Now it includes all Persons!

RECAP: Inner Joins

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
      JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

Inner Joins + NULLS = Lost data?

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)  
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store  
FROM    Product  
        JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store  
FROM    Product, Purchase  
WHERE   Product.name = Purchase.prodName
```

However: Products that never sold (with no Purchase tuple) will be lost!

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
 - I.e. If we join relations A and B on $a.X = b.X$, and there is an entry in A with $X=5$, but none in B with $X=5$...
 - A LEFT OUTER JOIN will return a tuple (a, NULL)!
- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store  
FROM   Product  
       LEFT OUTER JOIN Purchase ON  
           Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

INNER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
       INNER JOIN Purchase
       ON Product.name = Purchase.prodName
```

Note: another equivalent way to write an
INNER JOIN!



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Other Outer Joins

- Left outer join:
 - Include the left tuple even if there's no match
- Right outer join:
 - Include the right tuple even if there's no match
- Full outer join:
 - Include the both left and right tuples even if there's no match

Activity-3-3.ipynb

Summary

SQL is a rich programming language
that handles the way data is processed
declaratively